|  |  |
| --- | --- |
|  | Projet :Technologies Pour Web Service- REST WEB SERVICE |
|  |  |
|  | Majd Abou Akar (7052f) |
|  |  |
|  |  |
|  | SMB214 – REST WEB SERVICE  4/25/16 |

**Tables des matières**

[**Tables des matières** 1](#_Toc434925090)

[**Liste des abréviations** 2](#_Toc434925091)

[**Introduction** 3](#_Toc434925093)

[**Services REST / JSON** 4](#_Toc434925098)

[1. Introduction 4](#_Toc434925099)

[2. Architecture REST 4](#_Toc434925100)

[a. Principes directeurs 5](#_Toc434925101)

[b. Principes d'implémentation 7](#_Toc434925102)

[c. Représentation des ressources : JSON 8](#_Toc434925103)

d. Les Avantages Du REST……………………………………………………………………………………………………………….

[**Conclusion**](#_Toc434925119) 10

[**Références**](#_Toc434925120) 11

**Liste des abréviations**

|  |  |
| --- | --- |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| HTTPS | Hypertext Transfer Protocol Secure |
| JSON | JavaScript Object Notation |
| UI | User Interface |
| URI | Uniform Resource Identifier |
| REST | Representational State Transfer |
| EAI | Enterprise Application Integration |
| SOAP | Simple Object Access Protocol |
| SOA | Service Oriented Architecture |
| WSDL | Web Services Description Language |
| XML | Extensible Markup Language |
| XSLT | Extensible Stylesheet Language |

**Introduction**

De plus en plus d’entreprises ont besoin de rendre leurs applications accessibles sur le web. Les motivations sont multiples : élargir l’audience des utilisateurs, vendre des services en ligne, faire communiquer des applications existantes ou supporter une interface de type AJAX. Dans ces conditions, quelle architecture choisir ou concevoir ? Quel format utiliser pour échanger des données sur le web ? Et devrait-on utiliser un protocole applicatif existant ou développer un protocole adapté à nos besoins ?

Nous parlons ici de mettre en place un protocole applicatif sur le web (REST Web Service) pour échanger de l’information avec un grand nombre de clients. Une erreur de conception ou une faille de sécurité engendreraient des coûts exorbitants si elles devaient être corrigées. C’est une différence importante avec la problématique des EAI . Une erreur de conception sur une architecture interne a un coût mais peut être corrigée, le nombre d’utilisateurs étant limité et surtout connu. Il convient donc de se poser les bonnes questions si l’on veut réussir la migration de ses applications ou de son système d’information sur le web.

**Services REST / JSON**

# Introduction

Dans l’année 2000, Roy Fielding qui est un informaticien américain et l'un des principaux auteurs de la spécification HTTP, et le cofondateur du célèbre projet Apache HTTP Server, codifie l'architecture du Web dans sa thèse de doctorat intitulée « *styles architecturaux et la conception d'architectures logicielles basées sur les réseaux.* », il a introduit un style d'architecture connu sous le nom de Représentationnel State Transfer (REST), qui décrit la fondation du World Wide Web.

REST soit Représentationnel State Transfer , selon Wikipédia *« …est un style d'*[*architecture*](https://fr.wikipedia.org/wiki/Architecture_informatique)*pour les systèmes*[*hypermédia*](https://fr.wikipedia.org/wiki/Hyperm%C3%A9dia)*distribués..* » [[1]](#endnote-1), et est devenue l'une des technologies les plus importantes pour les applications Web et mobiles, dans l’actualité, il semble que toutes les technologies se dirigent vers une orientation d'API Web de type REST.

Un service basé sur REST est appelé un service RESTful, chaque langue majeure de développement inclut des cadres pour la création de services Web RESTful, nous pouvons trouver en PHP des cadres comme : [Slim:](http://www.slimframework.com/) , Epiphany, [Laravel](http://laravel.com/), etc, En Java : Jersey, Dropwizard, RESTEasy, Spring Rest, etc, en .Net : ASP Net Web API.

Ainsi les réseaux sociaux tels que Flickr, Twitter, Facebook, etc. sont basés sur REST, par exemple pour l’API Twitter [[2]](#endnote-2) nous pouvons demander l’information d’un utilisateur (Roy Fielding), avec le suivant url, si nous possédons l’authentification nécessaire.

[*https://api.twitter.com/1.1/users/show.json?screen\_name=royFielding*](https://api.twitter.com/1.1/users/show.json?screen_name=royFielding)

# Architecture REST

L'architecture REST a pris de plus en plus d'importance ces dernières années, comme nous avons déjà mentionné, nous retrouvons ce type d'architecture dans de nombreux services et réseaux sociaux. Il est de plus en plus difficile d'imaginer un service qui ne propose pas d'API avec laquelle interagir. REST décrit un style d'architecture logicielle permettant de construire une application devant fonctionner sur des systèmes distribués, typiquement internet.

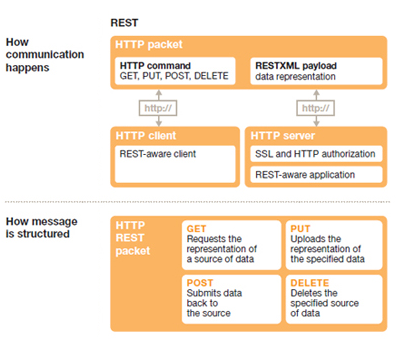


Figure 2 : Mécanisme de communication des services web RESTful.

*Source : http://www.pwc.com/us/en/technology-forecast/2012/issue2/features/feature-consumerization-apis.jhtml*

Dans l’architecture REST, toute information est une ressource et chacune d’elles est désignée par un URI, généralement un lien sur le Web. Les ressources sont manipulées par un ensemble d’opérations simples et bien définies. L’architecture client-serveur de REST est conçue pour utiliser un protocole de communication sans état le plus souvent HTTP. Avec REST, les clients et les serveurs échangent des représentations des ressources en utilisant une interface et un protocole bien définis. Ces principes encouragent la simplicité, la légèreté et l’efficacité des applications.

## Principes directeurs

* Architecture client/serveur : le serveur héberge le web service, et le client interroge ce serveur.
* Sans état : deux requêtes d'un client sont indépendantes, ce qui veut dire qu'au niveau serveur on ne traite pas une requête en référençant des éléments d'une requête précédente. Au niveau client, tout ce qui est nécessaire au traitement de la requête doit être inclus dans celle-ci. Au niveau HTTP, cela veut dire que l'on ne crée pas de session utilisateur dans laquelle on stocke des informations.
* Utilisation de mécanismes de cache possible, système en couche : l'idée est de pouvoir bénéficier du système distribué sous-jacent en permettant la mise en place de cache à chaque étape entre le client et le serveur. Pour HTTP, cela consiste essentiellement en l'utilisation de serveur proxy.
* Interface uniforme : c'est le point principal de différence par rapport aux Web Services : tout élément offert à la manipulation par l'application est nommé ressource et est identifié de manière unique.

HTTP définit les Identifiants de Ressource Uniforme (URI ci-après) suivant le schéma :

*http\_URL = "http:" "//" host [ ":" port ] [ abs\_path [ "?" query ]]*

Les différentes actions possibles sur ces ressources sont données par les différents types de requêtes HTTP, principalement GET, POST, PUT et DELETE. Nous manipulons des représentations des ressources, par les ressources directement. Les ressources sont donc encodées selon un format spécifique dans les messages HTTP.

## Principes d'implémentation

Pour définir une API REST, les étapes suivantes doivent être suivies :

• Définition des ressources manipulées, collection de ressources (liste) ou ressource unique.

• Codage de la représentation des ressources : quels sont les attributs d'une ressource, quel format va-t-on utiliser ?

• Sémantique des messages : les actions possibles sur les ressources sont indiquées par les messages du protocole de transport, ce qui donne pour HTTP :

* **GET** : récupération de la représentation d'une ressource ou d'une liste de ressource.
* **PUT** : mise à jour d'une ressource existante, création d'une ressource en spécifiant l'URI de la ressource.
* **POST** : création d'une sous ressource (le serveur décide de l'URI), ajout d'information à une ressource existante.
* **DELETE** : effacement.
* **HEAD** : informations sur une ressource (Entête de la requête HTTP).
* **OPTIONS** : permets de connaître les verbes autorisés pour une URI donnée. Une ressource donnée ne sera pas obligatoirement manipulable par tous les messages. Par exemple, une ressource accessible en lecture seulement peut n'être accessible que par les messages de type GET.

L'image ci-dessus illustre comment les composants interagissent dans le RESTful API design

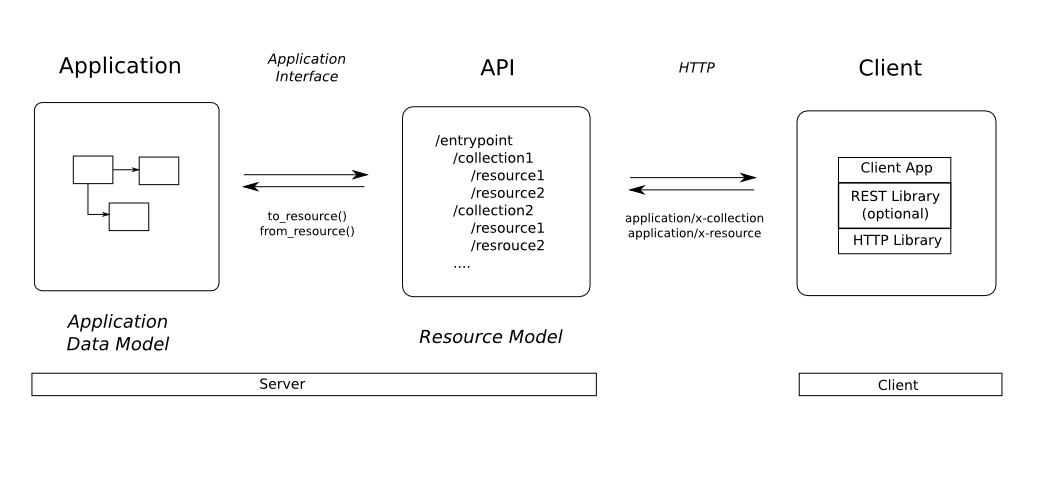


Figure 3 : Composants interagissent dans le RESTful API design.

*Source : http://restful-api-design.readthedocs.org/en/latest/scope.html*

## Représentation des ressources : JSON

Que ce soit au niveau du serveur ou au niveau des clients, une API REST manipule des ressources par une représentation de celles-ci. Le principe en pratique est de passer de la représentation utilisée en interne, que ce soit sur le client ou le serveur, à la représentation utilisée dans le message HTTP, requête ou réponse. Concernant la spécification du format de représentation utilisé dans le message : comme spécifié plus haut, les messages d'une application REST sont indépendants les uns des autres ce qui implique en particulier que le codage de la représentation des ressources peut être différent entre deux messages et donc qu'il faut spécifier dans le message le codage utilisé.

Pour un Web Service RESTful on utilise typiquement un entête HTTP :

*Content-type: text/xml*

Le client doit demander un format d'encodage spécifique en utilisant l'entête Accept:

*Accept: text/xml*

Pour les Web services RESTful, les encodages les plus utilisés sont XML et JSON. XML est un standard incontesté, mais souffre de quelques inconvénients :

* Verbeux.
* Difficilement lisible par un humain.
* Dualité entre les attributs et les éléments.

## Les Avantages Du REST

Nous avons vu que le respect du style d’architecture REST nous permettait d’avoir une interface simple et uniforme pour chacune des ressources.

•ressources identifiées par des URIs ;

•ensemble de méthodes réduit (GET, PUT, POST, DELETE, …)

•ressource manipulées par leur représentation

Cette interface uniforme nous permet-elle enfin de pouvoir parler d’interopérabilité ? Je vais vous décevoir, non, le style d’architecture REST ne nous permet pas de nous assurer que notre application sera interopérable. Pourquoi ? Tout simplement parce qu’il paraît aujourd’hui très difficile de développer des systèmes totalement interopérables. Chaque application se place dans un domaine métier précis et possède son vocabulaire. Mais le style d’architecture REST permet de simplifier l’intégration de nouvelles applications à notre système existant.

D’une part grâce au modèle d’adressage uniforme des ressources (URI). Tous les composants du système utilisent le même outil pour nommer les choses. Il est plus facile de les manipuler. Le paradigme d’appel de méthode à distance utilise le nom des méthodes métiers pour identifier le service sur le web. L’adressage est donc totalement spécifique au service.

Ensuite, l’utilisation d’un nombre réduit de méthodes permet aux clients de savoir, avant même de contacter le service, quelles méthodes sont disponibles. De plus, la sémantique de ces méthodes étant défini clairement. Si le client veut récupérer une ressource, il y a de forte chance que la méthode GET fonctionne. Un client SOAP ne pourra jamais deviner le nom des méthodes avant d’avoir consulter le contrat WSDL.

Enfin, le fait de ne pas spécifier le format des messages échangés permet de rendre indépendant l’évolution du langage (format des données) et du support de communication (protocole). Cette indépendance permet d’interconnecter plus facilement des composants qui n’ont à priori rien à voir ensemble. L’exemple le plus connu actuellement est celui de RSS. Un client peut tirer parti de l’information fournie par un flux RSS sans se soucier de la façon dont il a été généré.

En standardisant le protocole de communication, le style d’architecture REST permet de se concentrer sur la véritable source de couplage fort : le format de données échangées. Dans son article « SOAP, REST and Interopérabilité » Paul Prescod cite 4 technologies pouvant aider à la résolution de cette problématique :

•transformations XSLT ;

•l’héritage de schémas XML

•XML générés par des requêtes XML

•RDF, DAML+OIL et les technologies du web sémantique

**Conclusion**

Un Web Service RESTful est très simple à utiliser (partie cliente) et relativement simple à écrire (partie serveur) à partir du moment où l'on fait appel à des bibliothèques existantes. Le seul prérequis est la connaissance de HTTP.

Ce type d'architecture connait un grand succès en ce moment, confirme par l'intérêt d'acteurs tels que Google, Yahoo! Ou encore Amazon. Une des raisons principales tient à l'importance croissante des navigateurs en tant que plateforme d'exécution sur les postes clients fixes mais également sur les terminaux mobiles. Les applications web sont en train de passer d'un mode ≪ document ≫ (le serveur retourne une mise en forme du résultat) a un mode desktop (le client charge la partie interface de

l'application, les requetés au serveur ne font plus que charger les données) essentiellement grâce à l'utilisation d'Ajax. Dans ce contexte, fournir une interface RESTful a une application permet d'envisager une intégrations relativement aisée. Comme le cout technique est peu élevé je pense qu'il serait donc sage, que ce dans nos développements internes ou comme fonctionnalité a inclure lors d'un appel d'offres, nous nous assurions de la présence d'une interface REST.

L'investissement pour un développeur est relativement minime, notamment s'il possède déjà une expérience en programmation d'applications web assez bas niveau, proche de HTTP. Il existe pour le moment peu de ressources livresques sur le sujet si l'on excepte [6]. Par contre, de nombreuses ressources décrivant des API existantes sont disponibles par une recherche web notamment sur les sites de Google (http://code.google.com) et Yahoo (http://developer.yahoo.com).

**Références**

1. <https://fr.wikipedia.org/wiki/Representational_State_Transfer> [↑](#endnote-ref-1)
2. https://dev.twitter.com/rest/reference/get/users/show [↑](#endnote-ref-2)